To detect cycle in an Undirected Graph using DFS we follow below steps-

- Create an adjacency list if we don't already have one.
- Call the dfs method and mark current element as visited.
- In this algo similar to BFS we do not pass the Node alone, we also pass the parent node.
- Now, for each neighbor node we recursively call the DFS & return true if we get true from recursive dfs calls. In case, it's already visited, we just check if the neighbor node is parent node and if it's not we return true.

One important note is that for graphs with disconnected components we must call DFS for all elements/nodes if they are not visited.

Time Complexity-

O(N) + O(N+2E)

First O(N) comes from visiting all nodes, checking if they are visited or not.
O(N+2E) comes from DFS, we visit all nodes plus every nodes & we visit all edges twice since it's an undirected graph.

Space complexity-

O(N) recursion call stack & visited array/set.

```java
class Solution {
    // Function to detect cycle in an undirected graph.
    public boolean isCycle(int V, ArrayList<ArrayList<Integer>> adj) {
        // Code here
        boolean[] vis = new boolean[V];
        for (int node = 0; node < V; node++) {
            if (!vis[node] && dfs(node, -1, adj, vis))
                return true;
        }

        return false;
    }

    private boolean dfs(int node, int parent, ArrayList<ArrayList<Integer>> adj, boolean[] vis) {
        vis[node] = true;

        for (int ng: adj.get(node)) {
            if (ng == parent)
                continue;
            if (vis[ng] || dfs(ng, node, adj, vis))
                return true;
        }

        return false;
    }
}
```